XML Transformations Based on Logic Programming

Dietmar Seipel[1] and Klaus Prätor[2]

[1] University of Würzburg, Institute for Computer Science
Am Hubland, D – 97074 Würzburg, Germany
seipel@informatik.uni-wuerzburg.de

[2] Berlin–Brandenburg Academy of Sciences and the Humanities
Jägerstr. 22–23, D – 10117 Berlin, Germany
praetor@bbaw.de

Abstract. Critical or scientific editions are a promising field for the application of declarative programming, which can facilitate the parsing and the markup of texts, and the transformation of XML documents.
We have used a logic programming–based approach for the production of critical editions: In particular, we propose a transformation of XML documents based on a compact and intuitive substitution formalism. Moreover, we have developed a new XML update language FNUPDATE for adding navigation facilities.
We have implemented a transformation tool in SWI–PROLOG, which integrates and interleaves PROLOG's well–known definite clause grammars and the new substitution formalism.

Keywords. PROLOG, grammars, critical editions, XSLT

1 Introduction

The mainstream approach to XML processing is based on concepts such as the *Extensible Style Sheet Language for Transformations* (XSLT), the XML query language XQuery, and the underlying path language XPATH. Originally, XSLT was meant as an advanced sort of Cascading Style Sheets (CSS), but then it developed into a tool for the transformation of markup – the real formatting tool was delivered afterwards as XSL Formatting Objects (XSL–FO).

The way XSLT works resembles remarkably the way of PROLOG. The transformation is done by traversal of a tree, and by testing the matching of nodes against patterns in the style sheet. Comparing PROLOG and XSLT, of course the mainstream character and the extent of support are arguments in favour of XSLT, but there are also some drawbacks [9]. Firstly, XSLT is not a universal programming language, which means that not all problems are solvable within this framework. Secondly, XSLT is not designed from ground up as a declarative language. In simple examples this is not easily visible, but as applications get more complex you see the barely disguised imperative background shining through in control structures (xsl:if, xsl:for-each etc.) and pattern matching. A detailed comparison of XSLT and PROLOG can be found on the WWW pages of SWI–PROLOG [13].

Logic programming is very well–suited for natural language processing (NLP) [8] and for text processing in general. In particular, *definite clause grammars* (DCGs) are a powerful declarative approach to writing parsers [11]. According to Richard O'Keefe the great advantage of DCGs is that it is important *not* to think about the details of the translation: Any time you have stereotyped code, using a translator to automatically supply the fixed connection patterns means that the code is dramatically *shorter* and *clearer* than it would otherwise have been, because you are hiding the uninformative parts and revealing the informative parts. Thus, the code is *easier* to write and to read, and it has *fewer* mistakes, because the pattern is established once in the translator and thereafter tirelessly applied with machine consistency. DCGs are just one example for a more general idea (cf. [10, 14]): define a *little language* embedded in PROLOG syntax, write an interpreter for that language, and devise a translator which turns constructs of that language into the (useful, non–book–keeping) code that the interpreter *would* have executed for those constructs.

Prätor [15] has shown that techniques of logic programming are especially apt for the structures and the specific problems of critical editions. DCGs and a graph traversal had been used for transforming between the different layers of XML documents in the pilot project Jean Paul. In the present paper we extend this approach by proposing a more compact and intuitive substitution formalism, which can be interleaved with PROLOG's built–in DCG formalism, and we have developed a new XML update language FNUPDATE for adding navigation facilities.

The rest of this paper is organized as follows: In Section 2 we review some characteristics of critical editions in general, and we give an introduction to the different types of XML structures used within the pilot project Jean Paul. In Section 3 we show how XML documents can be represented in PROLOG, and we present transformations based on sequential scans and on tree traversals, respectively. Finally, in Section 4 we use an XML update language FNUPDATE, that we have implemented in PROLOG, for adding topology and navigation facilities to XML documents.

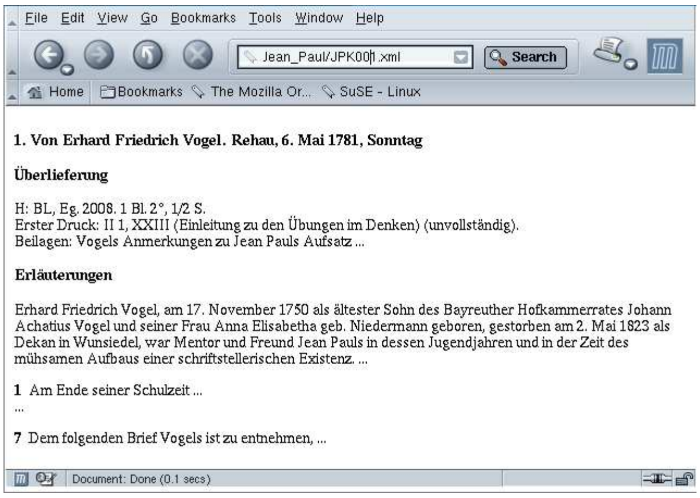
2 Critical Editions

Critical editions are notoriously difficult sorts of text, as they form not one linear text but rather a complex of different *variants* and readings of a text, which are to be handled within a critical apparatus (think of a set of foot– or endnotes). They are enriched by *commentaries* with historical and philological information and made accessible by indices and directories. Other peculiarities are the mostly large extent of the editions and the fact that the period of production as well as of usage is very long reaching from decades to centuries. Especially in electronic editions it is necessary to care for sustainable availability and usability.

Some problems are due to the print form. Lack of space leads to elaborate systems of abbreviations and to steady considerations, which material and information can still be incorporated and which one has to be left out. The print can hardly handle the inherent nonlinearity of the documents, and of course there is no thought of adaptation to different situations of usage. Compared to the print form, electronic editions show some advantages, which result from the advanced ways of navigation and retrieval and from the possibility to provide different types of output. The larger storage capacities provide room for additional information regarding, e.g., involved persons or historical circumstances. Hypertext capacities facilitate the constitution of temporal, spatial or thematic relations. Different editions may be nested and entirely new information spaces may be created in this way.

2.1 The Pilot Project Jean Paul

The Academy of Sciences and Humanities of Berlin–Brandenburg is the home of many edition projects of all ages – editions in a broader sense including aside from work editions also source editions and dictionaries. Most of these editions are still focussed on the print form, but we are working on the migration to genuine electronic editions.


File Edit View Go Bookmarks Tools Window Help
Jean_Paul/JPK001.xml
Search
Home
Bookmarks
The Mozilla Or...
SuSE - Linux
Document: Done (0.1 secs)

1. Von Erhard Friedrich Vogel. Rehau, 6. Mai 1781, Sonntag

Überlieferung

H: BL, Eg. 2008. 1 Bl. 2°, 1/2 S.
Erster Druck: II 1, XXIII (Einleitung zu den Übungen im Denken) (unvollständig).
Beilagen: Vogels Anmerkungen zu Jean Pauls Aufsatz ...

Erläuterungen

Erhard Friedrich Vogel, am 17. November 1750 als ältester Sohn des Bayreuther Hofkammerrates Johann Achatius Vogel und seiner Frau Anna Elisabetha geb. Niedermann geboren, gestorben am 2. Mai 1823 als Dekan in Wunsiedel, war Mentor und Freund Jean Pauls in dessen Jugendjahren und in der Zeit des mühsamen Aufbaus einer schriftstellerischen Existenz. ...

1 Am Ende seiner Schulzeit ...
...

7 Dem folgenden Brief Vogels ist zu entnehmen, ...

Fig. 1. Comment to a Letter of the Jean Paul Edition

Jean Paul has been a frequently read and appreciated author of novels in the times of Goethe. The Jean Paul Edition is organized in 6 volumes, each of which consists of about 150 letters to Jean Paul. The two aims of the pilot project were to produce an electronic equivalent of the just finished first volume, and to demonstrate some additional features of an electronic edition in an enriched selection. Aside from the usual contents it provides a simple full text search, and – most importantly – three paths to the letters via correspondent, year or place of the respective letter.

In this paper we focus on the *comments* to the letters. For each letter there exists one comment; the comment to the first letter of the first volume is shown in Figure 1.

The source texts of the edition were originally produced with Microsoft Word, which is not a desirable choice as an editor's tool at all, but just legacy. They were subsequently translated to a *migration layer* in HTML by a commercial tool. In the Sections 3 and 4 we will show transformations from this migration layer layer to the *archive layer* of the edition. The timeframe, in which a critical edition should be usable, is very long. Different users or situations may demand different presentations, which can then be produced from the archived documents.

2.2 Transformation Techniques

Two orthogonal concepts are used for transforming documents, DCGs and subsititution rules, which can also be mixed by calling them from each other:

- We use *DCGs* for *grouping* elements that are on the same level in the source document to form complex, nested structures. They are well–suited for parsing a sequence of items by a sequential scan.
- We use *subsititution rules* for transforming a complex, nested document. They *traverse* a tree–shaped XML document recursively: first the subelements of an element are transformed, then the resulting element is also transformed.

We propose a new, compact and intuitive substitution formalism, which can be interleaved with PROLOG's well–known built–in DCG formalism,

3 Transformation of XML Documents

The transformation of XML documents is a perfect task for declarative programming. The first success is to see how seamlessly an XML document converts into a genuine PROLOG structure. Subsequently, we show how to strip off unnecessary graphical markup and to transform layout markup which transports meaning into explicit XML tagging. This transforms from the migration to the archive layer.

3.1 XML Documents in PROLOG

Essentially, an XML or HTML element can be represented as a nested term structure containing the tag name, the attribute list, and the content (possibly also a list or empty). There are some libraries available to support this sort of transformation in both directions. As far as we know the first was the HTML tool P*i*LL*o*W for CIAO PROLOG [4], which also contains some CGI– and HTTP–support. A similar library for SGML exists for SWI–PROLOG.

The tool FNQUERY [16], which was implemented in SWI–PROLOG, uses a slightly abbreviated – but similar – term structure. E.g., an XML element

```
<p style="margin-top:0;margin-bottom:0;">
   <font face="Times New Roman" size="3">
      <em>1,</em> 18-19
      <strong>Gehen </strong> bis
      <strong>sind] </strong>
      Dem folgenden Brief Vogels ...
   </font>
</p>
```

is represented as

```
p:[style:'margin-top:0;margin-bottom:0;']:[
   font:[face:'Times New Roman', size:3]:[
      em:['1,'], '18-19',
      strong:['Gehen '], bis, strong:['sind] '],
      'Dem folgenden Brief Vogels ...' ] ]
```

called FN triple, where the attribute list can be omitted, if it is empty. Here, a triple is represented by means of operator definition in the form `T:As:C`, where `As` is an associative list of attribute/value pairs in the form `a:v`. Text elements (such as `'18-19'`) are simply represented as PROLOG atoms. The PROLOG library FNQUERY goes beyond the pure transformation by providing additional means to select and handle substructures, comparable to XQuery or F–logic. E.g., if `Xml` is the FN triple above, then the call `X := Xml^font@face` selects the value `'Times New Roman'` of the `face`–attribute of the nested `font`–element and assigns it to `X`. A detailed description of the selection features of FNQUERY can be found in [16].

3.2 Complex Transformations by Traversal of Documents

The predicate `transform_fn_item/2` transforms an FN triple `Item_1` (e.g., from the migration layer) into another FN triple `Item_2` (e.g., from the archive layer) based on facts for the predicate `--->/2`:

```
transform_fn_item(T:As:Es, Item) :-
   maplist( transform_fn_item,
      Es, Es_2 ),
   ( T:As:Es_2 ---> Item
   ; As = [],
     T:Es_2 ---> Item ).
transform_fn_item(declare(_X), '').
transform_fn_item(Item, Item).
```

This new *subsitution formalism* has been implemented within the subsystem FN-TRANSFORM of FNQUERY. It generalizes XSLT style sheets, which can be seen as a collection of transformation rules. These template rules in XSLT have the form of

```
<xsl:template match=Pattern>
   Template</xsl:template>
```

`Pattern` is an XPATH expression, which matches with some nodes of the document tree, and `Template` is the content which is to be inserted at this location. This content may contain the possibly recursive application of the same or other template rules. For doing this in PROLOG, we use substitution rules of the form

```
Item_1 ---> Item_2 :- Condition.
```

If a term matches the pattern `Item_1` (and of course all capacities of the PROLOG unification can be used to do this matching) and the call of `Condition` (which can be a standard PROLOG goal) succeeds, then the pattern will be replaced by the template `Item_2`.

3.3 From the Migration Layer to the Archive Layer

On the migration layer the text is represented as a *flat* sequence of paragraph elements, as shown in the previous subsection. The following *substitution rules* transform this sequence to a *nested* structure.

```
html:_:Es ---> Es.
head:_:_ ---> '' :- assert(commentHead).
body:_:Es_1 ---> comment:Es_2 :-
   jean_paul_comment(comment:Es_2, Es_1, []).

p:As:Es ---> signedp:Es :-
   right := As^align.
p:_:Es ---> notep:Es.

font:As:Es ---> lat:Es :-
   'small-caps' := As^'font-variant'.
font:_:[X] ---> X.
em:_:Es ---> commentHead:Es :- retract(commentHead).
em:_:Es ---> page:Es.

strong:_:Es ---> lemma:Es.
u:_:Es ---> spaced:Es.

T:As:Es ---> T:As:Es.
```

The body of the HTML document is transformed into a `comment`–element, and the first `em`–element in the body becomes the header of this comment; this is acchieved by the assertion of `commentHead`. Subsequent `em`–elements are transformed to `page`–elements. The following text is the result of the transformation. It is archived, and it serves as the basis for the production of different presentation layers. In the archive layer the comment consists of a header, which is a `commentHead`–element, and two blocks, which are given by `ednote`–elements with a `type`–attribute having the value `Überlieferung` and `Erläuterungen`, respectively.

```
<comment>
   <commentHead>1. Von Erhard Friedrich Vogel.
      Rehau, 6. Mai 1781, Sonntag
   </commentHead>
   <ednote type="Überlieferung">
      <notep>H: BL, Eg. 2008. 1 Bl. 2ř, 1/2 S.</notep>
      ...
   </ednote>
   <ednote type="Erläuterungen">
      ...
      <notep>
         <page>1,</page> 18-19
         <lemma>Gehen</lemma> bis <lemma>sind]</lemma>
         Dem folgenden Brief Vogels ist zu entnehmen, ...
      </notep>
   </ednote>
</comment>
```

The grouping of a comment into a header and two blocks – in the third substitution rule above – has been acchieved using the predicate `jean_paul_comment/3`, which is defined by the following DCG rules:

```
jean_paul_comment(comment:Es) -->
   header(H),
   block('Überlieferung', X),
   block('Erläuterungen', Y),
   { append(H, [X, Y], Es) }.

header([notep:[]:[commentHead:Es_1]|Es_2]) -->
   sequence_of_notep([notep:[]:[page:_:Es_1]|Es_2]).

block(Type, ednote:[type:Type]:Seq) -->
   sequence_of_notep([notep:_:[Type]|Seq]).

sequence_of_notep([notep:As:Es]) -->
   [notep:As:Es].
sequence_of_notep([notep:As:Es|Seq]) -->
   [notep:As:Es],
   sequence_of_notep(Seq).
```

According to the DCG rules, the header can consist of several `notep`–elements; the first of these elements is derived from an FN triple `notep:[]:[page:_:Es_1]` that is transformed to another FN triple of the form `notep:[]:[commentHead:Es_1]`. Using furher DCG rules the `notep`–elements of the archive layer can be transformed to `note`–elements, such as the following:

```
<note id="7">
   <remark>
      <position page="1" line="18-19"/>
      <lemma>Gehen</lemma> <lemma>sind</lemma>
   </remark>
   Dem folgenden Brief Vogels ist zu entnehmen, ...
</note>
```

The archive layer preserves all information, which may be helpful in some case, and omits all presentation specific features.

4 Update of XML Documents

In this section we investigate transformations that are applied to the archive layer for producing a presentation layer or a presentation layer with *navigation utilities*, respectively. The transformations are based on the XML update language FNUPDATE, which we have implemented in PROLOG within FNQUERY.

4.1 Pruning of XML Documents

Using FNUPDATE we delete the `remark`–elements within the `ednote`–elements for `Erläuterungen`, since they shall not be displayed at the presentation layer:

```
xml_extract_notes(Xml_1, Xml_2) :-
   Xml_2 := Xml_1 <-> [
      ^ednote::[@type='Erläuterungen']
      ^note^remark ].
```

In FNUPDATE an XML element given by an FN triple `Xml_1` can be pruned by deleting certain subelements. `<->` is a binary operator, which indicates that within the following pair of brackets it will be specified by a path expression which subelements should be deleted. In the example, certain `remark`–elements are deleted; the condition `@type='Erläuterungen'` assures that only `ednote`–elements with the value `Erläuterungen` in their `type`–attribute are affected.

The PROLOG predicate `xml_extract_notes/2` is applied to a `comment`–element `Xml_1`, and thus we don't have `comment` in the selection path. The result `Xml_2`, which forms the presentation layer, is also a `comment`–element; compared to `Xml_1` the `note`–elements are simplified; in our example we obtain

```
<note id="7">
   Dem folgenden Brief Vogels ist zu entnehmen, ...
</note>
```

as a simplified subelement of the presentation layer structure.

4.2 Topology and Navigation

The presentation layer is not meant as the graphical interface, which is primarily produced by the style sheet and the browser, but as an organisation of the content with regard to different user interests.

A very important task in this context is navigation. We suggest to distinguish between topology and navigation with reference to a document. *Topology* refers to the potential connections of document elements, whereas *navigation* means the realised and used connection paths. In accordance with this convention, providing the topology would be part of the archive layer, whereas the presentation has to care of the navigation. While topology is a matter of conceptual and structural relations, navigation has also to take into account technical and aesthetic aspects of the realisation. Other user interests demand different ways of access and navigation, but these should be generated from the one archive layer, or better: the archive layer should be designed in a way that multiple navigations can be produced with little effort.

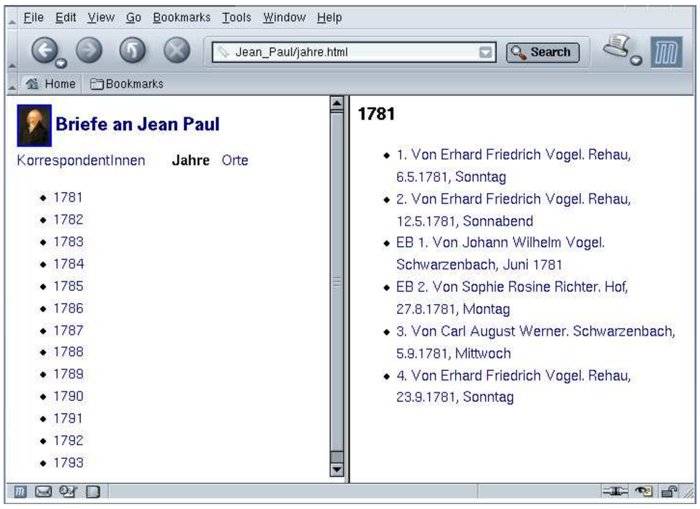
The information from the document, even in its tagged archive format, may not be enough to produce the navigation. In such case we need meta information, incorporated into the document (e.g., as RDF) or kept in a separate file or database. Here again PROLOG is a good choice to handle and inference these structures. The presentation layer may be seen as a virtual layer on top of documents and meta information. Derived from such meta information are both the navigation headlines of the letters and the different directories of persons, years and places, which provide access to the content, cf. Figure 2.

Another navigation task is the creation of indices, e.g. for persons mentioned in the document. For this purpose we would insert a durable tag that marks the person names, like `<person>Name</person>`, on the archive layer. This provides the basis for the insertion of the appropriate linking structures on the presentation layer.

Creating an Index for an XML *Document*

The following predicate adds anchors to a given word within an FN term using FNUPDATE: If a given word `Name` occurs in the content list of a `note`–element, then the word is replaced by an element `<person>Name</person>`:

```
xml_enrich_with_anchors(Name, Xml_1, Xml_2) :-
   Xml_2 := Xml_1 <-+> [
```



Briefe an Jean Paul

KorrespondentInnen **Jahre** Orte

- 1781
- 1782
- 1783
- 1784
- 1785
- 1786
- 1787
- 1788
- 1789
- 1790
- 1791
- 1792
- 1793

1781

- 1. Von Erhard Friedrich Vogel. Rehau, 6.5.1781, Sonntag
- 2. Von Erhard Friedrich Vogel. Rehau, 12.5.1781, Sonnabend
- EB 1. Von Johann Wilhelm Vogel. Schwarzenbach, Juni 1781
- EB 2. Von Sophie Rosine Richter. Hof, 27.8.1781, Montag
- 3. Von Carl August Werner. Schwarzenbach, 5.9.1781, Mittwoch
- 4. Von Erhard Friedrich Vogel. Rehau, 23.9.1781, Sonntag

Fig. 2. Navigation Utilities

```
      ^ednote^note^child::’*’::[
         ^self::’*’=N, name_contains(Name, N)]
      ^person:[N] ].
```

This is initiated by the binary replacement operator `<-+>/2` in the path expression. E.g., if we want to enrich an XML document containing the element

```
   <note id="7">
      Dem folgenden Brief Vogels ...
   </note>
```

with anchors to words containing `Vogel`, then we assume that the content of this element is split into tokens, i.e., that we have the following FN triple:

```
   note:[id:7]:[
      ’Dem’, folgenden, ’Brief’, ’Vogels’, ... ].
```

Using the FNUPDATE statement above we obtain the following:

```
   <note id="7">
      Dem folgenden Brief <person>Vogels</person> ...
   </note>
```

Finally, we create the index structure from the enriched document. For each inserted anchor, the following predicate extracts a corresponding reference of the form `<a href="#N">Name</a>` from the updated document, where the ID `N` is generated using the call `get_num(anchor_id, N)`:

```
xml_extract_references(Xml, index:Anchors) :-
   findall( a:[href:R]:[Name],
      ( [Name] := Xml^_^person^content::*,
        get_num(anchor_id, N),
        File := Xml@file,
        concat([File, '#', N], R) ),
      Anchors ).
```

Observe, that the selection `Xml^_^person` yields a complete `person`–element, such as `person:[]:['Vogels']`. The subsequent expression `^content::*` selects its content list. Thus, in the rule above `Name` would be assigned to `'Vogels'`.

Given a presentation layer file `letter_17` containing the enriched `note`–element above, we obtain an index with a reference to `letter_17`:

```
<index>
   <a href="letter_17#1">'Vogels'</a>
   ...
</index>
```

This index will be stored in a separate XML file.

5 Final Remarks

The introduction of the substitution predicate `--->/2` extends the ideas of the DCG operator `-->/2`, which is usually used for transformations on flat sequences of tokens, to transformations on complex structures based on tree traversals. Using `--->/2`, we have simplified the transformation formalism of [15].

We have used the sublanguage FNUPDATE of FNQUERY for a compact specification of updates to the archive layer of the critical Jean Paul edition. This was used for the construction of further navigation structures in addition to the hierarchical table of contents, cf. Figure 2, which we had before. Now we have a word register as well. In the future we want to base the creation of the whole navigation structure including the hierarchical table of contents on FNUPDATE and FNTRANSFORM.

At the moment the handling of meta data is a rather modest matter in our project. The next step will be to incorporate meta information in RDF and to get nearer to the *Semantic Web*. Without doubt this is a very interesting and important development for the future of critical editions. Examples of logic programming tools and libraries for RDF are to be found for example within SWI–PROLOG and within the Mozilla project [3]. As RDF is a format which must be supplemented by a separate means for inference, PROLOG (and companions) will be our favourite also in this field.

References

1. *S. Abiteboul, P. Bunemann, D. Suciu:* Data on the Web – From Relations to Semi–Structured Data and XML, Morgan Kaufmann, 2000.
2. *G. Antoniou, F. van Harmelen:* A Semantic Web Primer, MIT Press, 2004.
3. *D. Brickley:* Enabling Inference, `http://www.mozilla.org/rdf/doc/inference.html`.
4. *D. Cabeza, M. Hermenegildo:* WWW Programming using Computational Logic Systems (and the PiLLoW / CIAO Library), Proc. Workshop on Logic Programming and the WWW, at WWW6, 1997.
5. *S. Ceri, G. Gottlob, L. Tanca:* Logic Programming and Databases, Springer, 1990.
6. *M.A. Covington:* Natural Language Processing for Prolog Programmers, Prentice Hall, 1993.
7. *G. Gazdar, C. Mellish:* Natural Language Processing in PROLOG: An Introduction to Computational Linguistics, Addison–Wesley, 1989.
8. *G. Gupta, E. Pontelli, D. Ranjan, et al.:* Semantic Filtering: Logic Programming Killer Application, Proc. Symposium on Practical Aspects of Declarative Languages PADL 2002, Springer LNCS 2257.
9. *M. Leventhal:* XSL Considered Harmful, `www.xml.com/pub/a/1999/05/xsl/xslconsidered_1.html`.
10. *L.S. Levy:* Taming the Tiger – Software Engineering and Software Economics, Springer, 1987.
11. *R.A.O'Keefe:* The Craft of Prolog, MIT Press, 1990.
12. *C. Lehner:* PROLOG und Linguistik, Oldenbourg Verlag, 1990.
13. *B. Parsia:* Long story about using SWI–PROLOG, RDF and HTML Infrastructure, especially Chapter 6: DCGs Compared to XSLT, `http://www.xml.com/pub/a/2001/07/25/prologrdf.html`.
14. *F. Pereira, S. Sheiber:* PROLOG and Natural–Language Analysis, Center for the Study of Language and Information, 1987.
15. *K. Prätor:* Logic for Critical Editions, Proc. Intl. Conference on Applications of Declarative Programming and Knowledge Management INAP 2004.
16. *D. Seipel:* Processing XML Documents in PROLOG, Proc. 17th Workshop on Logic Programming WLP 2002.
17. *M. Smith, C. Welty, D. McGuinness:* OWL Web Ontology Language Guide, February 2004, `http://www.w3.org/TR/2004/REC-owl-guide-20040210/`.
18. *J. Wielemaker, A. Anjewierden:* Programming in XPCE/PROLOG `http://www.swi-prolog.org/`